Information Leakage as a Scheduling Resource

Fabrizio Biondi[1], Mounir Chadli[2], Thomas Given-Wilson[2], and Axel Legay[2]

[1] CentraleSupélec, France
[2] Inria, France

Abstract. High-security processes have to load confidential information into shared resources as part of their operation. This confidential information may be leaked (directly or indirectly) to low-security processes via the shared resource. This paper considers leakage from high-security to low-security processes from the perspective of scheduling. The workflow model is here extended to support preemption, security levels, and leakage. Formalization of leakage properties is then built upon this extended model, allowing formal reasoning about the security of schedulers. Several heuristics are presented in the form of compositional preprocessors and postprocessors as part of a more general scheduling approach. The effectiveness of such heuristics are evaluated experimentally, showing them to achieve significantly better schedulability than the state of the art. Modeling of leakage from cache attacks is presented as a case study.

1 Introduction

This paper considers a shared resource system where processes are classified as either *high-security* or *low-security*. High-security processes work with confidential information that should not be leaked to low-security processes. Typically, this includes loading confidential information into memory for use within high-security processes. Examples of such confidential information include encryption keys, medical data, and bank details. This confidential information may be vital to the operation of the high-security processes, but must also be tightly controlled and not be leaked to low-security processes. For instance, in an embedded sensor, high-security encryption processes handle encryption keys that must not be leaked to low-security data compression processes.

However, high-security processes may not properly flush confidential information from the shared resource, or context switching may interrupt their execution before such flushing can be applied. Consequently, confidential information remaining in the shared resource becomes (directly or indirectly) available to low-security processes.

Consider the small example in Fig. 1, written in Intel x86-64 assembly code for Linux compiled to ELF format[3]. There are two processes: `Process 1` doing some (trivial) encryption operations, and `Process 2` attempting to access the encryption key. `Process 1` takes a *key* `$KEY` and a *message* `$MSG` then encrypts the message with the key using an exclusive or `XOR` operation. The result is then output to the disk (represented by `$DISK1`). `Process 2` writes to a different disk location (represented by `$DISK2`) the content of register `r13`. It is clear that if `Process 2` is executed after the first operation and before the fourth operation of `Process 1`, then the value of the key is directly leaked.

```
; Process 1:
 mov    r13,$KEY   ; load key to register r13
 mov    r14,$MSG   ; load message to register r14
 xor    r14,r13    ; encrypt message with key
                   ; using XOR, store result in r14
 xor    r13,r13    ; wipe value of key
 out    $DISK1,r14 ; output the ciphertext (r14)

; Process 2:
 out    $DISK2,r13 ; output r13 (may store the key)
```

Fig. 1: Example Processes with schedule-dependent confidential information leakage.

If a scheduler is aware of a process' access level, then the scheduler can take action to prevent confidential information being leaked to low-security processes. Recent work [15, 17] has explored these kinds of problems in a real-time setting by scheduling a complete resource (memory) flush after any high-security process that is followed by a low-security process. However, this provides only limited options to the scheduler since such a complete resource flush is expensive and may prevent real-time tasks from meeting their deadlines. Further, when flushing is not possible, current approaches do not quantify the information leakage, simply considering any leakage unacceptable.

This paper proposes treating *confidentiality*, measured by the *resulting leakage* of secure information, as a quantitative resource that the scheduler can exploit. This allows for better quantification of the resulting leakage in different scenarios, as well as having a clear measure of the cost of different scheduling choices. Further, this allows for the creation of schedulers that can make better scheduling choices and also respect confidential information leakage constraints.

The paper builds upon the *workflow model* commonly used to represent real-time systems [3, 9, 26]. In the workflow model a set of *tasks* periodically produce *jobs* that have to be scheduled to complete before *deadlines*.

The workflow model is here extended by considering tasks to be composed of *steps*, each of which has an *execution time*, *leakage value*, and *security level*. Each one of these steps is implicitly an atomic sequence of actions that can be taken within a task without preemption by the scheduler. Thus a task consists of an ordered sequence of steps to be performed, that yields the total behavior of the task.

Using this extended workflow model, schedulers can operate upon steps rather than jobs, and so implement preemption while also being able to reason about leakage in a fine-grained manner. This supports offline schedulers in periodic systems that can plan an optimal strategy, as well as online schedulers that optimize using the knowledge available. (The focus in this paper is on the former.) Further, schedulers can be considered that operate over leakage thresholds or within quantified security constraints.

The approach in this paper easily captures prior work [15, 17] by inserting a *flush* task that has a known runtime cost and ensures a complete resource (memory) wipe

[3] Technical details for X86-64 (https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf) and ELF initialization (http://lxr.linux.no/linux+v3.2.4/arch/ia64/include/asm/elf.h).

(and thus zero resulting leakage). When a high-security job would be followed by a low-security job, a flush is inserted between them. This enforces zero resulting leakage, but often results in poor schedulability due to the high cost of frequently flushing resources.

By considering the flush task to be always available (rather than at prescribed times), schedulers can add flushes when this reduces resulting leakage and still achieve schedulability. Indeed, it is often possible to achieve zero resulting leakage even when flushing after every high-security job is not possible. Thus, solutions can be found here that achieve zero leakage that could not be scheduled by the prior state-of-the-art.

More generally, this paper proposes heuristic algorithms to achieve efficient scheduling while reducing resulting leakage, i.e. the amount of confidential information that can be leaked to low-security jobs. Thus allowing for more flexibility in choices; a scheduling approach may allow a limited amount of leakage to achieve schedulability. The scheduling algorithms presented here produce a schedule for a set of tasks. Standard scheduling algorithms are extended with a *preprocessing* and a *postprocessing* phase. Preprocessing modifies the set of tasks to be scheduled, while postprocessing modifies the schedule produced by a scheduling algorithm to yield another schedule. Several pre- and postprocessors designed to reduce leakage are introduced in this paper.

Experimental results are presented that demonstrate the trade-off between leakage and schedulability. These show that this approach schedules, with good (or zero) resulting leakage, sets of tasks that are not schedulable by the state-of-the-art. Different pre- and postprocessors and their impact on the resulting leakage are evaluated. This clearly illustrates that there is a trade-off to be made between leakage and schedulability. Accepting some leakage can allow for schedulability when requiring zero leakage would fail to be schedulable. Further, experimental results here show that zero leakage can still be achieved in cases where the current state-of-the-art fails schedulability.

A case study demonstrates the flexibility of the model, by detailing how to represent cache attacks and their leakage using the extended workflow model. This demonstrates a different leakage model and alternative ways to exploit the model.

Key Contributions. The key contributions of this paper are as follows.

- A model to reason quantitatively on the amount of information leaked by scheduling tasks with different security levels on a shared resource system.
- A scheduling approach with compositional and specialized pre- and postprocessors that schedule tasks while reducing the amount of confidential information leaked.
- Several heuristic pre- and postprocessing algorithms that can reduce leakage.
- Experimental evaluation of the combinations of the pre- and postprocessors, showing that the approach provides significantly better schedulability and lower information leakage than the state of the art.
- A case study showing how to adapt the model to other scenarios and kinds of leakage, demonstrated with cache attacks.

The structure of the paper is as follows. Section 2 recalls background information. Section 3 extends the workflow model. Section 4 presents our approach to scheduling used here, with algorithms for pre- and postprocessing. Section 5 highlights and discusses the experimental results. Section 6 presents a case study on adapting leak-

age for cache attacks. Section 7 discusses variations and extensions to the model and algorithms. Section 8 concludes and considers future work.

2 Background

Workflow Model. This section recalls the workflow model, a standard model for the scheduling of periodic tasks [3, 9, 26]. Section 3 extends the workflow model to account for the possible leakage of confidential information. Assume an infinite time divided into discrete time units indexed by natural numbers. Let Γ be a set of independent periodic *tasks* $\{\mathcal{T}_\alpha, \mathcal{T}_\beta, \dots\}$ with each task $\mathcal{T}_x \in \Gamma$ having a *period* P_x, an *execution time* E_x, and a *relative deadline* D_x. A *job* $\tau_{x,k}$ is produced by the activation of a task $\mathcal{T}_x \in \Gamma$ at *release time* $R_{x,k} = (k-1)P_x$, $\forall k \in \mathbb{N}_0$. Each job $\tau_{x,k}$ must be completed before its *absolute deadline* $A_{x,k} = R_{x,k} + D_x$. The *hyperperiod* H_Γ of a set of tasks Γ corresponds to the least common multiplier of the period P_x of each task $\mathcal{T}_x \in \Gamma$: $H_\Gamma = lcm\{P_x \mid \mathcal{T}_x \in \Gamma\}$.

Scheduling Algorithms. This paper uses two standard scheduling algorithms to schedule the jobs produced by sets of tasks: Earliest Deadline First (EDF) and Least Slack First (LSF). Both are simple and widely used offline scheduling algorithms based on dynamic priority of the jobs being scheduled. EDF determines the priority of jobs according to their absolute deadline. At any given point in time, out of the currently available jobs, the job with the earliest absolute deadline is scheduled first. LSF determines the priority of jobs according to their amount of *slack*. This slack is calculated for a job $\tau_{x,k}$ according to the formula $A_{x,k} - t - E_x$ where t is the current time. At any given point in time, out of the currently available jobs, the job with the least slack is scheduled first.

Information Leakage. *Information leakage* quantifies the amount of confidential information leaked by a system, and is widely used to measure of the (in)security of the system [1, 2, 4, 12]. In this paper, leakage is used to measure the amount of confidential information that a high-level job leaves in the shared resource at different moments of its execution. The unit of measure of leakage is not relevant since it depends on the specific application. For instance, if the confidential information is a private key, leakage could measure the number of bits of the key that are leaked. Alternatively, leakage could measure the number of confidential packets leaked from a secure transmission. Therefore, the same leakage model can be used with different leakage measures, where zero leakage represents no loss of confidential information.

Related Work. Real-time systems need to communicate with the outside world, such as receiving data from sensors or communicating with other systems, sometimes over unsecured networks. This communication has allowed attacks against even air-gapped industrial control systems [8].

The real-time scheduling requirement itself can be exploited to generate additional vulnerabilities. For instance, a process can modulate its use of a resource to affect the scheduling of another process, and use this to covertly transmit information [20, 21].

Further vulnerabilities can occur in any system with shared resources. When processes with different security levels share the same memory resources, it is possible for low-security processes to monitor the access to confidential information by high-security processes, causing information leakage [15]. Using separated memory for pro-

cesses with different security levels is expensive, particularly if the system has more than two security levels. Mohan et al. [15] consider a shared memory scenario where low-security processes executing after high-security processes could access the high-security processes' memory space resulting in information leakage. To prevent this, they propose completely flushing the shared resource (memory) after the execution of high-security processes when followed by a low-security process. In [17], Pellizzoni et al. generalize this work by introducing a binary relation NO-LEAK on tasks, where NO-LEAK($\mathcal{T}_x, \mathcal{T}_y$) holds if no leakage can occur from $\mathcal{T}_x$ to $\mathcal{T}_y$. The authors also determine the number of resource (memory) flushes needed to enforce the NO-LEAK relation, and consequently construct a preemptivity-assignment scheduling algorithm. This work proposes a more fine-grained approach to confidentiality in similar scenarios.

Another less formal approach is that used in [24] where they limit the time between preemptions between virtual machines in an online scheduling scenario to prevent cache attacks. This can be represented using the approach here as a case study.

Intel propose the *Software Guard Extension* (SGX) architecture to prevent leakage through shared memory [7]. SGX aims to keep each process in a separate enclave, and keep these enclaves isolated from other processes (and flushing them upon exit). However, Schwarz et al. [19] demonstrate that SGX is not safe using cache attacks.

Formal analysis of scheduling system under resource constraints has been performed by Kim et al. [13, 14]. The proposed approach can be extended to confidentiality as a resource using the model proposed in this paper.

3 Model

This section introduces the key concepts and model of the system being scheduled, and is based upon the workflow model recalled in Section 2. The extension here is to represent precise information about the internal operations and preemptivity of tasks by dividing them into *steps*. Steps include their own execution time (like a task or job), and are extended to include leakage value and security level. Special tasks are also added to model other operations of the system. The rest of this section details this extended model and presents illustrative examples that motivate the choices in this paper.

3.1 Concept

This section considers concepts and motivations for the model presented here; the division of tasks into steps, accounting for leakage, and justification for special tasks.

Steps. This model considers the possibility to divide tasks into fine-grained steps. A step represents an atomic sequence of operations that cannot be interrupted by preemption. The practical implementation of steps depends on the architecture and granularity of the scheduling system. The model is agnostic to step implementation details as long as an execution time, leakage value, and security level can be defined for each step. The most fine-grained approach would be to consider each CPU operation as a step. For instance, `Process 1` in Fig. 1 would be represented as a task divided into five steps. Thus, a task could be preempted after each CPU operation. Although very simple, in practice this approach is too fine-grained. In lightweight and embedded systems it is common

to delegate part of the handling of preemption and atomicity to the programmer, so it is reasonable to consider that the programmer themself could define the steps.

Special Tasks. This paper considers two special tasks representing special system operations: *flush* and *wait*. The flush task flushes all confidential information from the shared resource, for instance by overwriting all shared memory with zeroes. This preserves compatibility with the state of the art [15, 17] where flushing is used as the main tool to preserve confidentiality. The wait task represents idle processor time. Apart from the obvious use, scheduling of idle time can impact confidentiality of the system.

Leakage Values. The leakage value of a step represents the amount of confidential information that would be leaked to an attacker able to read the shared resource just after the steps' execution. The model does not constrain the way the leakage value is obtained: leakage can be added by the programmer as an annotation, computed by an automatic tool [5, 6, 23], or possibly both. For instance, the programmer could specify critical zones in which the program must not be interrupted, and the leakage values would be computed automatically by a tool (for both critical and non-critical zones). An alternative, variable-based approach would be to have the programmer annotate some variables as containing confidential information at a certain point (and as cleared of confidential information at a later point). Taint analysis can be used to identify which variables are tainted at each point. Information leakage quantification can be used to quantify leakage from the tainted variables.

3.2 Formal Model

Steps, Tasks and Jobs.

Definition 1 (Step). *Formally, each* step *is a tuple* $\mathcal{S}(E, L, X)$ *where* E *denotes the (worst case)* execution time *that the step takes to be completed,* L *denotes its (potential)* leakage value*, and* X *denotes its security level (either* high $\top$ *or* low $\bot$*).*

The (potential) leakage value L of a step $\mathcal{S}$ is a measure of the amount of confidential information left in a shared resource at the completion of $\mathcal{S}$. Here $\top$ indicates that the step contains confidential information and therefore is high-security. Similarly, $\bot$ indicates that the step should not have access to confidential information and therefore is low-security. Since $\top$ and $\bot$ are used to indicate whether the step has access to confidential information, $\bot$ steps typically have leakage zero. This is not a strict requirement, see Section 6. The choice of having two security levels here is to clearly illustrate the model, however the extension to any number of security levels is straightforward.

> For instance, consider `Process 1` in Fig. 1. Each assembly instruction can be represented by a single step with an execution time of one time unit and a security level of $\top$. The first three instructions have a leakage value of one, representing the fact that one word of confidential information (the key) is in the shared resource (in register `r13`). However, the remaining instructions have a leakage value of zero since the fourth instruction wipes `r13`.

The system operates with a set of *tasks* $\Gamma = \{\mathcal{T}_\alpha, \mathcal{T}_\beta, \ldots\}$.

Definition 2 (Task). *Each* task $\mathcal{T}_x \in \Gamma$ *is a tuple* $\mathcal{T}_x(P_x, D_x, \widehat{\mathcal{S}_x})$ *where* P_x *is the* period *of the task,* D_x *is its* relative deadline*, and* $\widehat{\mathcal{S}_x}$ *is a sequence of* steps $\mathcal{S}_{xa}, \mathcal{S}_{xb}, \ldots$ *making up the ordered actions of the task.*

Tasks are named with Greek letters, e.g. $\mathcal{T}_\beta$. Steps are named with the corresponding task's Greek letter and a Latin letter in alphabetical order, e.g. step $\mathcal{S}_{\beta c}$ represents the third step of task $\mathcal{T}_\beta$.

Observe that `Process 1` in Fig. 1 can be modeled by the following task:

$$\mathcal{T}_\alpha = \mathcal{T}(P_\alpha, D_\alpha, (\mathcal{S}_{\alpha a}(1,1,\top), \mathcal{S}_{\alpha b}(1,1,\top), \mathcal{S}_{\alpha c}(1,1,\top), \mathcal{S}_{\alpha d}(1,0,\top), \mathcal{S}_{\alpha e}(1,0,\top))).$$

Similarly, `Process 2` in Fig. 1 can be modeled by the following task:

$$\mathcal{T}_\beta = \mathcal{T}(P_\beta, D_\beta, \mathcal{S}_{\beta a}(1,0,\bot)) .$$

Definition 3 (Job). *Each* job $\tau_{x,k}$ *is created by the activation of the task* $\mathcal{T}_x$ *at* release time $R_{x,k} = (k-1)P_x$ *for* $k \in \mathbb{N}_0$*, and is a tuple* $\tau_{x,k}(R_{x,k}, A_{x,k}, \widehat{\mathcal{S}_{x,k}})$ *where* $A_{x,k} = R_{x,k} + D_x$ *is the job's* absolute deadline*, and* $\widehat{\mathcal{S}_{x,k}}$ *is the sequence of steps inherited from task* $\mathcal{T}_x$.

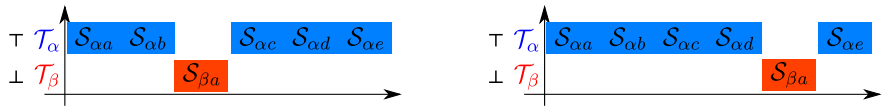
Jobs are named with the corresponding task's Greek letter and the number k, so job $\tau_{\beta 4}$ is the fourth job generated by task $\mathcal{T}_\beta$ and step $\mathcal{S}_{\beta 4c}$ is the third step of job $\tau_{\beta 4}$.

For simplicity, a task (resp. job) will be referred to as $\top$ or $\bot$ when all steps within that task (resp. job) are either $\top$ or $\bot$, respectively.

Flush and Wait. The model uses a task to represent complete flushing of the shared resource. The *flush* task is defined by $\mathcal{T}_\mathcal{F}(-,-,\mathcal{S}_\mathcal{F}(E_\mathcal{F}, 0, \top))$ where $E_\mathcal{F}$ is the execution time to completely flush the shared resource. Observe that *after flushing the shared resource the leakage is reduced to zero*. This is achieved by the single step $\mathcal{S}_\mathcal{F}(E_\mathcal{F}, 0, \top)$ that takes all the execution time of the flush task and has a zero leakage value. Since the flush task is always available to be scheduled, it has no defined period or deadline (denoted here as -), being able to scheduled (or not) at whim. The security level of flush is $\top$ since it is acceptable for flush to have access to confidential information, and for use in calculating the resulting leakage (see below). For simplicity and when no ambiguity may occur, $\mathcal{F}$ is used for the flush task or step.

To represent idle processor time, define the *wait* task as $\mathcal{T}_\mathcal{W}(-,-,\mathcal{S}_\mathcal{W}(1,*,*))$. Similar to flush, wait is always available to be scheduled and has no period or deadline (again denoted as -). Wait also has a single step that has the minimal runtime of one time unit. However, the leakage value of wait is here denoted by $*$ since waiting does not change the shared resource, instead the $*$ denotes that *the leakage value of a wait step is the same as the previous step*. Similarly, the security level is also represented by $*$ because it is the same as the previous step. Again for simplicity and where no ambiguity may occur, $\mathcal{W}$ may be used in place of the wait task or step.

Traces, Solutions, and Resulting Leakage.



(a) Solution leaking information. (b) Solution leaking no information.

Fig. 2: Schedulings for the processes in Fig. 1.

Definition 4 (Trace). *A* trace $\widetilde{\mathcal{S}} = (\mathcal{S}_1(E_1, L_1, X_1), \mathcal{S}_2(E_2, L_2, X_2), \dots)$ *is a (possibly infinite) sequence of* $n \in \mathbb{N} \cup \{\infty\}$ *steps that may come from any number of jobs.*

In a trace, Step $\mathcal{S}_1$ starts execution at time $t_1 = 0$, and each step $\mathcal{S}_i$ for $i > 1$ starts execution at time $t_i = \sum_{j=1}^{i-1} E_j$ and terminates execution at time $t_i + E_i$. The notation $\widetilde{\mathcal{S}_1} +\!+ \widetilde{\mathcal{S}_2}$ is used to indicate concatenation of traces $\widetilde{\mathcal{S}_1}$ and $\widetilde{\mathcal{S}_2}$, and $\widetilde{\mathcal{S}} \setminus \mathcal{S}_1$ the removal of the step $\mathcal{S}_1$ from the trace $\widetilde{\mathcal{S}}$. The focus of this paper is upon *solutions*.

Definition 5 (Solution). *A trace* $\widetilde{\mathcal{S}}$ *is a* solution $\overline{\mathcal{S}}$ *if:*

1. *for each job* $\tau(R, A, \widehat{\mathcal{S}})$*:*
   (a) *each step in* $\widehat{\mathcal{S}}$ *appears in the trace* $\widetilde{\mathcal{S}}$ *in the order that it appears in* $\widehat{\mathcal{S}}$*;*
   (b) *the first step of* $\widehat{\mathcal{S}}$ *does not start execution before* R*;*
   (c) *the last step of* $\widehat{\mathcal{S}}$ *does not terminate execution after* A*;*
2. *each step that is not wait* $\mathcal{W}$ *or flush* $\mathcal{F}$ *appears exactly once in the trace* $\widetilde{\mathcal{S}}$*.*

Given a set of tasks Γ, a solution $\overline{\mathcal{S}}$ is a *solution for* Γ, written $\overline{\mathcal{S}_\Gamma}$, iff $\forall \mathcal{T}_x \in \Gamma, \forall k \in \mathbb{N}_0$ then for each job $\tau_{x,k}(R_{x,k}, A_{x,k}, \widehat{\mathcal{S}_{x,k}})$ it holds that every step in $\widehat{\mathcal{S}_{x,k}}$ is in $\overline{\mathcal{S}}$.

A solution $\overline{\mathcal{S}}$ is *periodic* if it periodically repeats the same sequence of steps up to job indexing. For simplicity, a periodic solution may be represented by the periodically repeated sequence alone.

Given a trace $\widetilde{\mathcal{S}}$ the *resulting leakage* $\mathscr{L}(\widetilde{\mathcal{S}})$ of trace $\widetilde{\mathcal{S}}$ represents the total amount of information leaked during the execution of the jobs scheduled according to $\widetilde{\mathcal{S}}$.

Definition 6 (Resulting leakage). *Given a trace* $\widetilde{\mathcal{S}}$ *composed of* n *steps with* $n \in \mathbb{N} \cup \{\infty\}$*, the* resulting leakage $\mathscr{L}(\widetilde{\mathcal{S}})$ *of the trace* $\widetilde{\mathcal{S}}$ *is defined inductively as follows:*

- *if* $n \leq 1$*, then* $\mathscr{L}(\widetilde{\mathcal{S}}) = 0$*;*
- *if* $n > 1$ *and the second step* $\mathcal{S}_2$ *of trace* $\widetilde{\mathcal{S}}$ *is* $\top$*, then the resulting leakage is the leakage of the trace without the first step* $\mathcal{S}_1$*:* $\mathscr{L}(\widetilde{\mathcal{S}}) = \mathscr{L}(\widetilde{\mathcal{S}} \setminus \mathcal{S}_1)$;
- *if* $n > 1$ *and the second step* $\mathcal{S}_2$ *of trace* $\widetilde{\mathcal{S}}$ *is* $\bot$*, then the resulting leakage is the leakage of the trace without the first step* $\mathcal{S}_1 = \mathcal{S}(E_1, L_1, X_1)$ *plus the leakage value* L_1 *of the first step* $\mathcal{S}_1$*:* $\mathscr{L}(\widetilde{\mathcal{S}}) = \mathscr{L}(\widetilde{\mathcal{S}} \setminus \mathcal{S}_1) + L_1$.

Since every solution $\overline{\mathcal{S}}$ is a trace $\widetilde{\mathcal{S}}$, a solution's resulting leakage $\mathscr{L}(\overline{\mathcal{S}})$ is defined in the same manner.

Recall the example from Fig. 1. The solution in Fig. 2a has resulting leakage one, since `Process 2` is executed when the key is in the shared resource and so the step $\mathcal{S}_{\beta a}$ is able to access the key.

However, the solution in Fig. 2b has resulting leakage is zero, since `Process 2` is executed after the key has been wiped from the shared resource.

If a solution is periodic, the *periodic leakage* can be calculated as follows. Given one instance of the periodically repeated sequence of steps $\widetilde{\mathcal{S}} = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_i)$, the periodic leakage is the resulting leakage of the sequence $\widetilde{\mathcal{S}}{+}{+}\mathcal{S}_1$.

4 Our Approach

The overarching goal of the approach proposed in this paper is to produce a solution with low resulting leakage for a given set of tasks. To achieve this, standard offline scheduling algorithms are extended with a *preprocessing* and a *postprocessing* phase. The preprocessing phase transforms a set of tasks Γ into a set of preprocessed tasks Γ'. Then scheduling is applied to Γ' obtaining a solution $\overline{\mathcal{S}'}_{\Gamma'}$ for Γ'. Finally, the postprocessing phase transforms the solution $\overline{\mathcal{S}'}_{\Gamma'}$ into a postprocessed solution $\overline{\mathcal{S}''}_{\Gamma'}$. Both the pre- and postprocessing phases can affect the desired solution $\overline{\mathcal{S}''}_{\Gamma'}$, here with the goal of reducing the resulting leakage. The rest of this section presents various heuristic algorithms used for the results (see Section 5). The scheduling algorithms considered are EDF and LSF. Note that EDF and LSF do no consider the security-level or leakage of the steps (for discussion of this see Section 7). The rest of this section focuses upon the pre- and postprocessors. The division in phases creates a modular and compositional approach, allowing for a better comparison of different pre- and postprocessors.

4.1 Preprocessing

Preprocessors are algorithms that take a set of tasks Γ and produce a set of tasks Γ' to be scheduled. This paper considers preprocessors that attempt to "merge" adjacent steps with the same security level within each task in Γ. The merged step has the sum of the execution times of the merged steps, the leakage value of the last merged step, and the same security level as the merged steps. For instance, the steps $\mathcal{S}_{\alpha a}(1, 0, \top)$ and $\mathcal{S}_{\alpha b}(1, 4, \top)$ could be merged producing the step $\mathcal{S}_{\alpha a'}(2, 4, \top)$. The rest of this section presents three preprocessing algorithms that exploit merging.

Total Merge. The Total Merge algorithm merges all the steps in a task into a single step.The merging is achieved by starting with a step that has execution time and leakage value zero. The execution time for each other step in the task is then added, and the leakage value from the last step being merged is preserved. The security level is set to that of the last step (this is reasonable here since all steps within a task share the same security level, for other approaches to this see Section 6). Finally, the processed task uses this single merged step as its only step.

One-Step Merge. The One-Step Merge algorithm attempts to merge pairs of adjacent steps. Adjacent pairs are merged iff the leakage of the former step is higher than the latter. This is achieved by iterating through the steps $\mathcal{S}_i$ of the task. If $L_i > L_{i+1}$, then the steps $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ are merged. Otherwise, $\mathcal{S}_i$ is maintained unchanged. This algorithm generates a new sequence of steps $\widehat{\mathcal{S}'}$, that are then used in the processed task.
***n*-Step Merge.** A straightforward extension to the One-Step Merge algorithm is to allow merging of any number of steps. This appears in the results as *n*-Step Merge.

4.2 Postprocessing

Postprocessing algorithms take one solution and produce another solution. This can be done by any possible manipulation of the steps within the original solution $\overline{\mathcal{S}'}_\Gamma$ to produce the new solution $\overline{\mathcal{S}''}_\Gamma$ that does not break the property of being a solution for Γ. The rest of this section presents four such postprocessors.

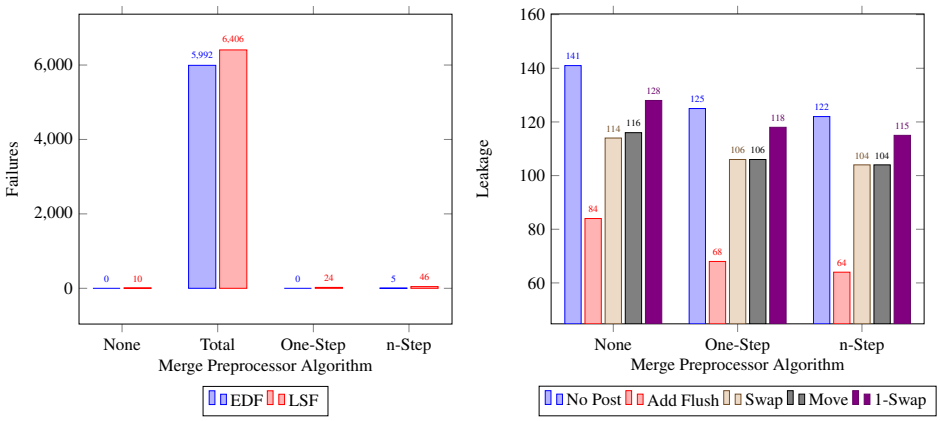
Add Flush. The Add Flush algorithm replaces sequences of $\mathcal{W}$ with $\mathcal{F}$ where possible. Add Flush operates by finding sequences of $\mathcal{W}$ whose length is greater than or equal the execution time of $\mathcal{F}$. If such a sequence is found, a $\mathcal{F}$ is added to the produced solution instead of the initial sequence of $\mathcal{W}$ with execution time equal to the $\mathcal{F}$. Any remaining $\mathcal{W}$ in the solution are maintained.
Swap. The Swap algorithm attempts to reduce the resulting leakage by swapping steps within the solution.Swap works by considering each step $\mathcal{S}_i$. Then each possible swap $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]$ between the step $\mathcal{S}_i$ and a following step $\mathcal{S}_j$ is considered. If the trace with this swap applied has less resulting leakage and is still a solution, then this solution $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ is kept as the best possible solution so far. Finally, once all possible swaps have been considered, the best swap to the solution is applied and i is incremented.
Move. The Move algorithm moves one step to a new position in the solution. Move works in the same manner as the Swap postprocessor, except instead of swapping $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ the steps $\mathcal{S}_i$ and $\mathcal{S}_j$, the move $[\mathcal{S}_i \longrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ moves the step $\mathcal{S}_i$ to be after $\mathcal{S}_j$. For example: $[\mathcal{S}_1 \longrightarrow \mathcal{S}_3]\mathcal{S}_a, \mathcal{S}_b, \mathcal{S}_c = \mathcal{S}_b, \mathcal{S}_c, \mathcal{S}_a$ where the first step $\mathcal{S}_a$ is moved to be after the third step $\mathcal{S}_c$. The rest of the algorithm is the same as Swap, finding the best possible move and ensuring the trace after the move is a solution. The algorithm is identical to the Swap algorithm substituting $[\mathcal{S}_i \leftrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ with $[\mathcal{S}_i \longrightarrow \mathcal{S}_j]\overline{\mathcal{S}}$ in Line 4.
1-Swap. Observe that if only swapping or moving with the following step is considered, that is $[\mathcal{S}_i \leftrightarrow \mathcal{S}_{i+1}]$ or $[\mathcal{S}_i \longrightarrow \mathcal{S}_{i+1}]$, then the swap and move postprocessors coincide. This postprocessor is denoted as 1-Swap in the results.

5 Experimental Results

This section discusses the results obtained by running experiments with the preprocessing, scheduling, and postprocessing algorithms in this paper.

The experiments were conducted by using approximately 30,000 randomly generated sets of tasks[4], and then testing each possible combination of one preprocessing, one

[4] 30,000 sets of tasks were generated, 22 were discarded as unschedulable.



(a) Number of failures for each combination of preprocessor and scheduling algorithm, out of ~30,000 experiments. Note that Total Merge, corresponding to the state of the art [15, 17], fails ~20% of the time.

(b) Information leakage of the solutions for each combination of pre- and postprocessor (except Total Merge) using the EDF scheduling algorithm.

Fig. 3: Failures and leakage results for pre- and postprocessing.

scheduling, and one postprocessing algorithm. Each set of tasks consists of 2 to 6 tasks with at least one $\top$ task and one $\bot$ task, with each task having 1 to 8 steps, and each step execution time from 1 to 5. Sets of tasks with a hyperperiod over 5000 have been discarded to reduce testing time. The code[5] to perform the tests and implement the preprocessing, scheduling, and postprocessing is written in Java 1.8, and all experiments conducted on a Linux 3.13 64-bit kernel on an Intel Core i7-3720QM 2.60GHz CPU with 8GB of RAM. A demo[6] is available that shows examples, and allows users to conduct their own GUI-based experiments. The rest of this section discusses experimental outcomes.

Preprocessor	**Postprocessor**				
Merge	**None**	**Add Flush**	**Swap**	**Move**	**1-Swap**
None	2	116	1919	1903	190
One-Step	1	93	1567	1489	149
n-Step	1	88	1486	1404	141

Table 1: Average execution time (in ms) for each combination of pre- and postprocessor (except Total Merge) using the EDF scheduling algorithm.

The first point of interest is the schedulability of the set of tasks used in each experiment. Merging task steps in a preprocessor can make a set of tasks unschedulable, and the EDF and LSF scheduling algorithms are not equally able to find solutions. The failure percentage for each combination of preprocessing and scheduling algorithm is shown in Fig. 3a.

[5] Available via git from: https://scm.gforge.inria.fr/anonscm/git/secleakpublic/secleakpublic.git
[6] Demo available via website at: http://secleakpublic.gforge.inria.fr/

Fig. 3a clearly shows that greater merging of steps leads to more schedulability failures. In particular, indicating that Total Merge is not an effective algorithm to use in practice despite being considered the current state of the art [15, 17]. This is a strong motivation for the approach presented in this work to consider fine-grained preprocessing and preemption of tasks. Due to its high failure rate, Total Merge will not be considered further in this paper.

Fig. 3a also shows that, for all preprocessing algorithms, EDF performs better for schedulability than LSF. (This is expected since EDF is guaranteed to find a solution if the tasks are schedulable, while LSF is not.) The two scheduling algorithms produce almost the same results for every other measure tested, so the rest of this paper shall present only experimental results using the EDF scheduling algorithm.

Comparing the experimental results from postprocessing algorithms, the average resulting leakages for each combination of pre- and postprocessor is shown in Fig. 3b, while the average running times to generate a solution are shown in Table 1.

As expected, solutions without any postprocessing produce the highest resulting leakage. The best resulting leakage is obtained by the Add Flush algorithm. (This would correspond to the approach in [15, 17] when combined with Total Merge, however as noted above this is often not schedulable.) Note that merging preprocessors reduce total time, since they reduce the number of steps that the scheduler has to schedule.

1-Swap slightly reduces the resulting leakage, however Table 1 shows that it is significantly more expensive than the scheduling operation, so 1-Swap could be applied after Add Flush only if the cost is acceptable. Swap and Move do not reduce the resulting leakage significantly more than 1-Swap and are significantly more expensive to compute. These indicate that there is a balance to be found depending on the scenario. Taking significant time to pre-compute an optimal scheduling strategy for a sensor or other real-time system prior to shipping could be worth the time cost. However, for online scheduling with limited (or no) ability to look ahead and consider such options, the cost of anything more complex than Add Flush or 1-Swap may be too much.

6 Case Study: Modeling Cache Attacks

This section demonstrates how to reason about cache attacks using the model presented in this paper, and how leakage can be used in different ways. This includes how to adapt resulting leakage to represent leakage via cache attacks, and how to exploit the general definition of the model to handle more complex notions of leakage.

In cache attacks, the shared resource is the cache itself. There are several approaches to gaining information from the cache (which is in general a form of side-channel attack) [10, 18, 22, 25]. One such method is for the attacker to attempt to load code that uses the same cache lines as the program being attacked. When these load very quickly, then this indicates that the program being attacked has already loaded particular parts of the program, and from this the attacker can infer information about the program.

The main point in modeling cache attacks is that leakage is related to the cache lines, and there are many such lines in the cache. Thus, the measure of leakage is which lines of the cache are known to have been loaded by the attacker.

This can be modeled using the techniques in this paper, by exploiting the flexibility of the leakage representation as follows:

- The leakage value L of a step is represented by a bit-vector, with 1 bit for each cache line. Loading a cache line is represented by setting the bit in the bit-vector that represents that line of the cache to 1.
- When calculating leakage from a trace, the leakage is calculated by taking the bit-wise disjunction (represented as |) of the leakage bit-vectors. Observe that this automatically accounts for over-writing by newer lines.
- The leakage result from a high security step to a low security step can then be calculated over bit-vectors, e.g. by the bit-wise conjunction operation (represented as &). Recall that since the attacker loads cache lines to test if another process has accessed these lines, they will appear to have loaded these lines to another step.
- The flush step $\mathcal{F}$ is represented by setting the leakage vector to 0000.

For example, assume four cache lines, then leakage would be represented by bit-vectors of length four. A step that loads into the first cache line would have the leakage bit-vector 1000, and the step that loads into the third cache line would have the leakage bit-vector 0010. If these steps were executed sequentially, the leakage bit-vectors 1000 and 0010 bit-wise disjoined would yield $1000 \,|\, 0010 = 1010$.

An attacker that attacks (by using) the first and second cache line would have a bit-vector 1100. If the leakage of the last high security step is 1010 and the attacker has leakage bit-vector 1100, then the attacker would gain information about the first cache line being used (since $1010 \,\&\, 1100 = 1000$), and the leakage would end up in the state 1110 (since $1010 \,|\, 1100 = 1110$) since the attacker must access these cache lines to perform the attack.

The leakage value calculated from the cache attacks can also be more realistic. In practice certain cache lines yield more information. So if the cache attack is being modeled for an attack against the key of AES [16, 22], different lines can be given different values, thus allowing precise computation of key leakage. Indeed, works such as [11] could be used to determine the most appropriate leakage values to use.

Thus, the model presented in this paper already supports many interesting and real scenarios by instantiating the leakage in an appropriate manner. This has been kept simple earlier in the paper for illustration, but highly complex leakage models can easily be accounted for in the manner demonstrated above.

7 Discussion

On the Division of Scheduling into Three Phases. The division into three phases is to separate out distinct parts of an overall scheduling from tasks to a solution. This approach allows for separation conceptually of different phases, and also for composition of simple algorithms in the pre- and postprocessing phases. For example, a postprocessor could move steps in a solution around to maximize contiguous $\mathcal{W}$s and then be

composed with the Add Flush postprocessor to improve the resulting leakage further. This also allows different strategies to be employed in different phases, including strategies with different goals. For example, processors for resulting leakage minimization and energy consumption could be combined during pre- or postprocessing (or both).

Online Scheduling. This paper considers offline scheduling, i.e. when the tasks to be scheduled are known beforehand. In most real cases the tasks appear at runtime, requiring online heuristics to decide the scheduling. The division in steps and the leakage model presented in this paper extend immediately to the online scenario. While the preprocessors and postprocessors do not, they provide insight that can be used to build online heuristics that reduce leakage. We consider this as future work.

Execution Time. This paper has considered the execution time to be essentially fixed for each step. Although formally the execution time is worst case, the scheduling here does not exploit when steps may terminate prior to their (worst case) execution time. This could naturally be incorporated into online scheduling (above), but even in a purely offline scheduling system this could be exploited. For example, consider the cache attack scenario, where flushing not only effects the leakage, but by flushing the cache the execution time will go up due to cache misses.

8 Conclusions and Future Work

In a system with shared resources, the security of confidential information is a major concern. This paper allows reasoning about leakage of confidential information by extending the workflow model to support fine-grained preemption and confidentiality. This allows confidentiality to be addressed by quantifying the amount of information leaked by the system, including different leakage models.

Scheduling in this new model is then considered using pre-and postprocessors. These can be compositonally combined for scheduling that exploits different techniques and approaches, including focusing on different aspects of the overall problem. Several pre- and postprocessing heuristic algorithms are presented that can operate on the model. These are focused upon improving resulting leakage, but the principles can be adapted to other areas as well. Experimental results evaluate the algorithms presented here, showing that the model and heuristics improve over the state of the art and show that even simple heuristics can be effective. The case demonstrates the flexibility of the model, and illustrates how to adapt to different kinds of leakage and scenarios.

Future work could generalise to multi-resource approaches, where scheduling considers confidentiality, energy consumption, schedulability, etc. Another direction would be to consider theoretical complexity, and optimal scheduling strategies.

References

1. M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In S. Chong, editor, *CSF*. IEEE, 2012.
2. M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *S&P*, pages 141–153. IEEE, 2009.

3. A. Benoit, U. V. Çatalyürek, Y. Robert, and E. Saule. A survey of pipelined workflow scheduling: Models and algorithms. *ACM Comput. Surv.*, 45(4):50:1–50:36, Aug. 2013.
4. F. Biondi, A. Legay, P. Malacaria, and A. Wasowski. Quantifying information leakage of randomized protocols. *Theor. Comput. Sci.*, 597:62–87, 2015.
5. F. Biondi, A. Legay, L. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 702–707. Springer, 2013.
6. T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. In *ESORICS*, pages 219–236, 2014.
7. V. Costan and S. Devadas. Intel sgx explained. *IACR ePrint Archive*, 2016:86, 2016.
8. N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier, 2011.
9. R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, Nov 1966.
10. D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, pages 279–299, 2016.
11. D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Usenix Security*, volume 15, pages 897–912, 2015.
12. J. Heusser and P. Malacaria. Quantifying information leaks in software. In C. Gates, M. Franz, and J. P. McDermott, editors, *ACSAC*, pages 261–269. ACM, 2010.
13. J. H. Kim, A. Legay, K. G. Larsen, M. Mikucionis, and B. Nielsen. Resource-parameterized timing analysis of real-time systems. In *HVC*, pages 190–205, 2015.
14. J. H. Kim, A. Legay, L. Traonouez, A. Boudjadar, U. Nyman, K. G. Larsen, I. Lee, and J. Choi. Optimizing the resource requirements of hierarchical scheduling systems. *SIGBED Review*, 13(3):41–48, 2016.
15. S. Mohan, M. Yoon, R. Pellizzoni, and R. Bobba. Real-time systems security through scheduler constraints. In *ECRTS*, pages 129–140. IEEE Computer Society, 2014.
16. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.
17. R. Pellizzoni, N. Paryab, M. Yoon, S. Bak, S. Mohan, and R. Bobba. A generalized model for preventing information leakage in hard real-time systems. In *RTAS*. IEEE, 2015.
18. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS '09*. ACM, 2009.
19. M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. *arXiv preprint arXiv:1702.08719*, 2017.
20. J. Son and J. Alves-Foss. Covert timing channel capacity of rate monotonic real-time scheduling algorithm in MLS systems. In *IASTED*, pages 13–18, 2006.
21. S. H. Son, R. Mukkamala, and R. David. Integrating security and real-time requirements using covert channel capacity. *IEEE Trans. Knowl. Data Eng.*, 12(6):865–879, 2000.
22. E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.
23. C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu. Precisely measuring quantitative information flow: 10k lines of code and beyond. In *EuroS&P*. IEEE, 2016.
24. V. Varadarajan, T. Ristenpart, and M. M. Swift. Scheduler-based defenses against cross-vm side-channels. In *Usenix Security*, pages 687–702, 2014.
25. Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.
26. M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *RTAS*, pages 1–12. IEEE, 2016.